Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing

Seyong Lee
Oak Ridge National Laboratory
lees2@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
Georgia Institute of Technology
vetter@computer.org

***Abstract*—Graphics Processing Unit (GPU)-based parallel computer architectures have shown increased popularity as a building block for high performance computing, and possibly for future Exascale computing. However, their programming complexity remains as a major hurdle for their widespread adoption. To provide better abstractions for programming GPU architectures, researchers and vendors have proposed several directive-based GPU programming models. These directive-based models provide different levels of abstraction, and required different levels of programming effort to port and optimize applications. Understanding these differences among these new models provides valuable insights on their applicability and performance potential. In this paper, we evaluate existing directive-based models by porting thirteen application kernels from various scientific domains to use CUDA GPUs, which, in turn, allows us to identify important issues in the functionality, scalability, tunability, and debuggability of the existing models. Our evaluation shows that directive-based models can achieve reasonable performance, compared to hand-written GPU codes.**

I. Introduction

Hardware accelerators, such as Graphics Processing Units (GPUs), have emerged as promising alternatives for high-performance computing, due to their unique combination of energy-efficiency, performance, density, and cost [1]. In particular, GPUs have been highlighted as possible building blocks for the future Exascale computing architectures, where a very limited power budget forces designs that include heterogeneous cores consisting of around a million nodes, where each node contains a heterogeneous and hierarchical massively parallel processor (MPP) having O(1000) cores. Many experts expect that the architectures and execution models similar to the current GPU computing may be suitable for the intra-node computing on these Exascale systems [2], [3]. However, the programming complexity of the GPUs still poses significant challenges for developers of efficient GPU codes. The complex interactions among the limited hardware resources on existing systems make it more difficult to orchestrate and optimize applications. Hence, developing productive GPU programming models is a crucial step toward addressing Exascale programming challenge, especially for intra-node programming.

Not surprisingly, there has been growing research and industry interest in lowering the barrier of programming these devices [4], [5], [6], [7]. Even though the Compute Unified Device Architecture (CUDA) programming model [8] and Open Computing Language (OpenCL) [9] offers a more user-friendly interface, programming GPUs is still complex and error-prone, when compared to programming general-purpose CPUs and using existing parallel programming models, such as OpenMP [10]. Most recently, several directive-based, GPU programming models have been proposed from both the research community (hiCUDA [11], OpenMPC [12], etc.) and industry (PGI Accelerator [13], HMPP [14], R-Stream [15], OpenACC [16], OpenMP for Accelerators [17], etc.). On the surface, these models appear to offer different levels of abstraction, and expected programming effort for code restructuring and optimization. In this rapidly changing area, there have been initial studies and benchmarking efforts that compare GPU directive models; these include Hernandez et al. [18] who evaluated both PGI Accelerator and HMPP by porting and evaluating two application kernels. Their work focused primarily on the optimization process to achieve a comparable performance to CUDA.

For more comprehensive understanding of the differences in these new models, in this work, we have ported and optimized thirteen OpenMP programs to CUDA GPUs using the models. These programs are two kernel benchmarks (*JACOBI* and *SPMUL*), three NAS OpenMP Parallel Benchmarks (*EP*, *CG*, and *FT*), and eight Rodinia Benchmarks [19] (*BACKPROP*, *BFS*, *CFD*, *SRAD*, *HOTSPOT*, *KMEANS*, *LUD*, and *NW*). They include both regular and irregular applications from diverse domains, such as Medical Imaging, Bioinformatics, Fluid Dynamics, Data Mining, etc. In our evaluation using these benchmarks, we captured salient results for applicability, functionality, performance, and optimization effort.

This paper makes the following contributions. (i) We present the first comprehensive evaluation of existing directive-based GPU programming models (PGI Accelerator, HMPP, OpenMPC, R-Stream, OpenACC) from both industry and academia. (ii) We have estimated the effectiveness of the directive-based GPU programming models by porting 13 OpenMP programs from diverse application domains to CUDA GPUs, which allows both qualitative and quantitative analysis of the GPU models, with respect to their applicability, complexity, and performance. (iii) We identify various issues and future directions that stimulate further research to push the state of art in productive GPU programming models.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

The rest of this paper is organized as follows: Section II provides an overview of the GPU architecture, and section III presents directive-based GPU programming models. Evaluation methodology, experimental results, and discussions are described in Section IV, Section V, and Section VI, respectively, and the conclusion will be presented in Section VII.

II. Overview of GPU Architecture and CUDA Programming Model

GPUs are general-purpose multi-threaded single instruction, multiple data (SIMD) architectures that are configured as a set of multiprocessors, each of which contains a set of SIMD processing units. In GPU programming models, such as CUDA and OpenCL, a GPU is viewed as a parallel computing coprocessor attached to a host CPU, and it can execute a large number of threads concurrently. A GPU program consists of a series of sequential and parallel regions. Sequential regions have little or no parallelism; they are executed on the CPU as serial codes. Parallel regions containing abundant parallelism are implemented as a set of *kernel functions*, which are executed on the GPU by a set of threads in an SIMD manner.

In the CUDA model, GPU threads are grouped as a grid of thread blocks, each of which is mapped to a streaming multiprocessor (SM) in the GPU. Threads within each thread block are mapped to SIMD processing units in the SM. The configuration of GPU threads are specified through language extensions at each kernel invocation.

The CUDA programming model assumes separate address spaces for a host CPU and a GPU device. The CUDA model provides a rich set of APIs for explicit GPU memory management, including functions to transfer data between the CPU and the GPU.

The CUDA memory model has an off-chip *global memory* space, which is accessible by all threads, a fast on-chip *shared memory* space, which is shared only by threads in the same thread block, an off-chip *local memory* space, and *registers*, which are private to each thread.

CUDA and other GPU models support multiple levels of parallelism, but each parallelism differs in its synchronization support, which often sets a limit on the types of applications that can exploit the abundant computing powers in GPUs.

III. Directive-Based, High-Level GPU Programming Models

General directive-based programming systems consist of directives, library routines, and designated compilers. In the directive-based GPU programming models, a set of directives are used to augment information available to the designated compilers, such as guidance on mapping of loops onto GPU and data sharing rules. The most important advantage of using directive-based GPU programming models is that they provide very high-level abstraction on GPU programming, since the designated compiler hides most of the complex details specific to the underlying GPU architectures. Another benefit is that the directive approaches make it easy to do incremental parallelization of applications, like OpenMP, such that a user can specify regions of a host program to be offloaded to a GPU device in an incremental way, and then the compiler automatically creates corresponding host+device programs.

There exist several directive-based GPU programming models [11], [12], [13], [14], [15], [16], [17]. These models provide different levels of abstraction, and programming efforts required to conform to their models and optimize the performance also vary. To understand the level of the abstraction that each directive-based model gives, Table I summarizes the type of information that GPU directives can provide. The table says that R-Stream offers the highest abstraction among the compared models, since in the R-Stream model, the compiler covers most features implicitly, without a programmer's involvement. The table also implies that hiCUDA seems to provide the lowest level of abstraction among the tested models, since programmers should control most of the features explicitly. However, lower level of abstraction is not always bad, since low level of abstraction may allow enough control over various optimizations and the features specific to the underlying GPUs to achieve optimal performance. Moreover, actual programming efforts required to use each model may not be directly related to the level of abstraction that the model offers, and high-level abstraction of a model sometimes puts limits on its application coverage. For better understanding of these issues, the following subsections give more detailed overview of each model that is evaluated in this paper.

A. PGI Accelerator

1) Programming Model: The PGI Accelerator programming model [13] is a directive-based model targeting general hardware accelerators, even though it currently supports only CUDA GPUs. The PGI model allows very high-level abstraction similar to OpenMP; at the minimum level, the user just has to specify regions, called *Accelerator Compute Regions*, where loops will be compiled into accelerator kernels. The user can do this by inserting directives into the host program without any additional modification on the original code structures; all the other details, such as actual accelerator-kernel-code generation, accelerator initialization, data transfers between a host and an accelerator, or accelerator startup and shutdown, are handled by the PGI Accelerator compiler. The PGI model also allows users to provide additional information to the compilers, including specification of data local to an accelerator region, bounds of accessed arrays, guidance on mapping of loops onto an accelerator, and so on.

The PGI Accelerator directives can be categorized as two types: directives for managing parallelism and those for managing data. The directives for parallelism guide types of parallelism to execute loops, and the ones for data deal with data traffic between the host and the accelerator. One noble feature in the PGI Accelerator model is the *data region*. The data region sets boundaries where data are moved between the host and the accelerator; if a single data region encloses many compute regions, the compute regions can reuse the data already allocated on the accelerator. This can dramatically

TABLE I

FEATURE TABLE, WHICH SHOWS THE TYPE OF INFORMATION THAT GPU DIRECTIVES CAN PROVIDE. IN THE TABLE, *explicit* MEANS THAT DIRECTIVES EXIST TO CONTROL THE FEATURE EXPLICITLY, *implicit* INDICATES THAT THE COMPILER IMPLICITLY HANDLES THE FEATURE, *indirect* SHOWS THAT USERS CAN INDIRECTLY CONTROL THE COMPILER FOR THE FEATURE, AND *imp-dep* MEANS THAT THE FEATURE IS IMPLEMENTATION-DEPENDENT.

Features		**PGI**	**OpenACC**	**HMPP**	**OpenMPC**	**hiCUDA**	**R-Stream**
Code regions to be offloaded		loops	structured blocks	loops	structured blocks	structured blocks	loops
Loop mapping		parallel vector	parallel vector	parallel	parallel	parallel	parallel
Data management	GPU memory allocation and free	explicit implicit	explicit implicit	explicit implicit	explicit implicit	explicit	implicit
	Data movement between CPU and GPU	explicit implicit	explicit implicit	explicit implicit	explicit implicit	explicit	implicit
Compiler optimizations	Loop transformations	implicit	imp-dep	explicit	explicit		implicit
	Data management optimizations	explicit implicit	imp-dep	explicit implicit	explicit implicit	implicit	implicit
GPU-specific features	Thread batching	indirect implicit	indirect implicit	explicit implicit	explicit implicit	explicit	explicit implicit
	Utilization of special memories	indirect implicit	indirect imp-dep	explicit	explicit implicit	explicit	implicit

reduce the overhead of data movement and is important for optimized performance.

2) Limitations: The PGI Accelerator model provides a powerful tool for managing data traffic between the host and the accelerator; using data regions properly annotated with data clauses, programmers can easily generate optimized data communication patterns. However, there was a practical limit in using data regions; data regions had to lexically contain compute regions. If those compute regions were in different procedures, they had to be inlined to get the benefit. This limit is lifted by using new directives such as *reflected* introduced for PGI Accelerator Fortran but still not fully implemented in PGI Accelerator C. The lexical scope issue exists in other parts too; if target loops have function calls, they can not be transformed into accelerator kernels, unless called functions are simple enough to be automatically inlined by the compiler.

The main limitation of the PGI model is that it does not provide enough control over various optimizations and translations; most of the optimizations are performed by the compiler automatically, but programmers have little control over them. Another limitation is that it does not provide directives for architecture-specific features. For example, CUDA GPU provides various special memories, such as shared, constant, and texture memories. Each memory has distinct features, and thus careful exploitation of these resources may be crucial for optimal performance. However, current PGI model does not provide tools to control data mapping onto these memories.

Lack of reduction clause may be also problematic. The compiler can detect simple scalar reduction patterns automatically, but if the reduction pattern is complex, the compiler either fails to detect or generates wrong output codes. Proper annotation of reduction patterns will be able to simplify the automatic reduction code generation.

There are several implementation-specific limits; there is a limit on the maximum depth of nested loops that the compiler can handle, and pointer arithmetic is not allowed within loops to be offloaded to the accelerator.

B. OpenACC

1) Programming Model: OpenACC [16], which is initiated by a consortium of CAPS, CRAY, PGI, and NVIDIA, is the first standardization effort toward a directive-based, general accelerator programming model portable across device types and compiler vendors. In this paper, we tested PGI's implementation of OpenACC. The OpenACC programming model is largely based on the PGI Accelerator programming model (cf. §III-A1), and thus it inherits most of important concepts in the PGI Accelerator model, such as *data region* and *compute region*. Like PGI Accelerator, OpenACC has two types of directives: directives for managing parallelism and directives for managing data. However, the OpenACC directives are further extended to express additional features not available in the PGI Accelerator model. First, OpenACC has two types of compute constructs: the *kernels* construct and *parallel* construct. The *kernels* construct defines a region whose execution behavior is very similar to the compute region in the PGI Accelerator model; the code in the *kernels* region will be divided by the compiler into a sequence of kernels, such that each loop nest becomes a single kernel. The *parallel* construct is new to OpenACC. It also defines a compute region, but its behavior is more similar to OpenMP's parallel region; the whole region will be compiled into a single kernel, which may contain one or more work-sharing loops. Second, OpenACC defines three levels of parallelism; *gang*, *worker*, and *vector*, while PGI Accelerator defines two levels: *parallel* and *vector*. Third, OpenACC has rich data clauses to optimize data movement across procedure boundary in more general ways than PGI Accelerator, applicable to both dummy arguments and global data without an explicit interface. Last, OpenACC has an explicit reduction clause for loops, while PGI Accelerator depends on the compiler to detect reduction operations implicitly.

2) Limitations: Because the OpenACC model is based on the PGI Accelerator model, it has limits similar to those in the PGI model; the OpenACC model does not provide enough

control over various optimizations and translations, and it also does not offer directives to express architecture-specific features. One OpenACC-specific limit is that OpenACC requires that data in a data clause should be contiguous in memory. In other words, if a multi-dimensional array consists of multiple, discontinuous chunks of data, programmers should change the memory layout by manually packing the arrays in their application's memory allocation code. At the time of writing this paper, the only available OpenACC implementation is built on top of the PGI Accelerator compiler, and thus the tested OpenACC compiler also has the same limits as the PGI Accelerator compiler. Because there exist several features not explicitly defined in the OpenACC specification, such as ambiguity in handling nested loops where gang/worker/vector loops are mixed, it is expected that porting OpenACC programs from one vendor compiler to another may incur performance portability and other code translation issues.

C. HMPP

1) Programming Model: HMPP [14] is another directive-based, GPU programming model targeting both CUDA and OpenCL. It also provides very high-level abstraction on GPU programming, similar to PGI Accelerator. HMPP model is based on the concept of *codelets*, functions that can be remotely executed on hardware accelerators like GPUs. A codelet is a pure function, which does not contain static or volatile variable declarations nor refer to any global variables except if these have been declared by a HMPP directive *resident*. The codelet does not return any value, and thus the result of the computation should be passed by some parameters given by reference. Because the codelet is a base unit containing computations to be offloaded to GPUs, porting existing applications using HMPP often requires manual modification of code structures, such as outlining of a code region or re-factoring existing functions. To minimize these additional code changes, HMPP also provides an alternative way, called *region*, with which programmers can annotate code blocks without outlining to a codelet. However, the same constraints for writing codelets are applied to regions. For optimized data management, HMPP uses the concept of *a group of codelets*. By grouping a set of codelets, data in the GPU memory can be shared among different codelets, without any additional data transfers between CPU and GPU. Combining the codelet grouping with HMPP *advancedload/delegatedstore* directives allows the same data-transfer optimizations as the *data region* in the PGI model.

One unique feature in the HMPP model is that the same codelet can be optimized differently depending on its call sites. The *codelet/callsite* directives can specify the use of a codelet at a given point of a program, such that related data transfers and a target device can be changed according to the calling contexts. HMPP also provides a rich set of directives, called *HMPP Codelet Generator Directives*, which allow programmers high-level control over CUDA-specific features, such as CUDA special memories, in addition to complex loop mapping and loop transformations.

2) Limitations: The concept of the HMPP codelet is unique; it defaults to a "buffer" memory mode, where GPU memory is allocated per codelet parameter, but not per host variable accessed by GPU, as in other GPU programming models. Therefore, in the HMPP codelet model, if two different host variables are passed to the same codelet parameter at different call sites, the two host variables will share the same GPU memory. This GPU memory sharing is good for saving GPU memory, but it may cause additional data transfers between CPU and GPU if the two host variables are passed to the same codelet in an interleaved manner, since later codelet call will overwrite previous data in the GPU memory. Another inefficiency occurs if the same host variable is passed to different codelets; in this case, separate GPU memory regions are allocated for each codelet, even though both codelets access the same host variable. To remove redundancy caused by duplicate GPU memory allocations, codelet parameters referring to the same host variable should be explicitly mapped using *map* or *mapbyname* directives, but this makes data sharing across codelets very verbose and error-prone. The *data mirroring* directives provide a simpler way to allocate GPU memory per host variable but not per codelet parameter. Even with the data mirroring directives, however, manual data-transfer optimizations in the HMPP model is still more verbose than other models, since the HMPP codelet model requires users to individually specify data-transfer rules for each codelet. In the PGI Accelerator/OpenACC models, on the contrary, data transfers for multiple compute regions can be controlled by one data region enclosing those regions.

In the HMPP model, compiler optimizations are explicit in that they should be explicitly specified by users using a rich set of directives. However, the current HMPP compiler implementation implicitly performs some optimizations for scalar variables, but if these optimizations conflict with user's manual optimizations, it will incur unexpected errors.

D. OpenMPC

1) Programming Model: OpenMPC [12] is a programming system that uses a variant of OpenMP extended for CUDA GPU programming. OpenMPC consists of a standard OpenMP API plus a new set of directives and environment variables to control important CUDA-related parameters and optimizations. Because OpenMPC is based on OpenMP, it provides programmers with the same level of abstraction as OpenMP, and most of existing OpenMP programs can be ported to CUDA GPUs seamlessly. In OpenMPC, OpenMP parallel regions are the base candidates to be executed on the GPU, but the OpenMPC compiler may split each parallel region into two sub-regions at each explicit/implicit synchronization point, to enforce OpenMP synchronization semantics under the CUDA programming model. Among the resulting sub-regions, the ones containing at least one OpenMP work-sharing construct will be transformed to GPU kernels. OpenMPC does not have directives like PGI *data region* to set arbitrary boundaries where GPU memory is allocated and freed. Instead, users can set the whole program or each function as an implicit

boundary, using an OpenMPC environment variable, and then the compiler automatically optimizes the data management using context-sensitive, interprocedural analyses. For context-sensitive optimizations, the OpenMPC compiler performs automatic, selective procedure cloning, which has an effect similar to what the HMPP *codelet/callsite* directives offer. OpenMPC also provides assorted data directives to control data transfer optimizations.

The main advantage of using OpenMPC is that it allows fine-grained control over various compiler optimizations, execution configurations, and CUDA-specific features such as special CUDA on-chip memories. Combined with various built-in tuning tools, the rich set of directives enable an easy tuning environment that assists users in generating CUDA programs in many optimization variants without detailed knowledge of the complex GPU programing and memory models. One of OpenMPC's unique features is that it supports both scalar and array reduction computations. Standard OpenMP has *reduction* clauses for scalar reduction operations. To express array reduction operations in OpenMP, the *critical* directives are commonly used. The OpenMPC compiler can automatically detect array reduction patterns in the OpenMP critical sections and transform them into appropriate GPU reduction codes. Additionally, reduction clauses in OpenMPC accept both scalar and array reduction variables.

2) Limitations: The OpenMPC model is extended from the OpenMP model. However, not all OpenMP semantics can be correctly preserved in the CUDA programming model, due to the architectural differences between traditional shared memory multiprocessors, served by OpenMP, and stream architectures, adopted by GPUs. Code regions containing OpenMP features not implementable in the CUDA model will be executed on the host. Splitting OpenMP parallel regions at every synchronization point is an essential step to identify code regions suitable for execution on the GPU. However, the splitting may break the correctness of a program, if the splitting results in *upward exposed* private variables. The compiler automatically performs various transformations to remove this problem, but there are cases that the automatic transformations cannot fix. In these cases, the compiler will notify the problems, and programmers should manually restructure the code regions to avoid the problems.

In the OpenMPC memory model, both OpenMP *shared* and *threadprivate* data are mapped to GPU global memory. However, current OpenMPC implementation allows only scalar or array types for these data. Converting a pointer-type variable to an array-type often requires outlining of OpenMP parallel regions. The burden of outlining will be removed if OpenMPC data clauses are extended such that programmers can specify the bounds of accessed data, as in other directive models.

The OpenMPC compiler provides a variety of powerful interprocedural analyses, which enable advanced optimizations without manual inlining. However, some of the aggressive optimizations may not be safe, since they rely on array-name-only analyses. The compiler does not guarantee the correctness of those optimizations, and thus users should check the correctness of the generated output codes. For this case, the compiler provides users with options to selectively apply these unsafe, aggressive optimizations.

Another implementation-specific limit is that the current OpenMPC compiler can handle a multi-dimensional array correctly only if it is allocated as one continuous data layout, which constrains the way a user can allocate multi-dimensional dynamic arrays.

E. R-Stream

1) Programming Model: R-Stream [15] is a polyhedral model-based, high-level, architecture-independent programming model, which targets various architectures, such as STI Cell, SMP with OpenMP directives, Tilera, and CUDA GPUs. The R-Stream model is based on a polyhedral compiler that can automatically parallelize mappable regions of code that conform to the *extended static control program* structure, which refers to a program that operates on dense matrices and arrays and that consists of *for* loops, whose bounds are integer affine functions, and arrays, whose indexing functions are also affine. In the R-Stream model, a user just has to specify which functions are valid mappable regions by tagging them with directives, and then the underlying compiler performs various complex loop analyses and transformations to automatically parallelize and optimize the mappable sequential codes for a target architecture.

The R-Stream model is distinguished from other directive-based approaches in several ways. First, in the R-Stream model, the parallelization of the code is fully automatic; its powerful polyhedral compiler automatically captures dependency in the input code and performs necessary loop transformations to maximize the parallelism, while other existing approaches are semi-automatic in that they require the programmer to explicitly specify necessary directives to guide the parallelization of the code either directly or indirectly. Second, R-Stream allows fully automatic management of the data transfers between the host and GPU and GPU-specific memories, such as CUDA shared memory. For this, the compiler performs hierarchical decomposition of the algorithm between and across host, multiple GPUs, and within-GPU, using hierarchical tiling. On the other hand, other works, except for OpenMPC, heavily rely on directives inserted by the programmer. OpenMPC provides automatic optimizations for both the memory transfers between the CPU and GPU, and for using GPU-specific memory, but it lacks tiling transformations as powerful as R-Stream. Third, R-Stream provides transparent porting across different architectures, while other approaches target only hardware accelerators, such as GPUs, and may not be performance portable. In R-Stream, the computation choreography needed for efficient execution on target architectures can be automatically constructed from the same sequential C source, and automatically rendered into the corresponding target-specific sources, such as OpenMP, CUDA, etc. This transparent mapping is possible due to a user-provided, declarative machine model, which provides the overall architecture topology as well as performance and capacity parameters; the

compiler uses the contents of the machine model to derive various optimization parameters and mapping tactics.

2) Limitations: The R-Stream's polyhedral framework provides an unprecedented level of loop transformations for more powerful parallelization and mapping capability than other existing approaches. However, it works only on regular programs with integer affine loop bounds and integer affine loop indexing, which are often too restrictive to be applied to real applications, as shown in the experimental results in Section V. There exist many regular scientific applications containing affine loops, each of which may be efficiently optimized by the R-Stream compiler. When porting non-trivial scientific applications into GPUs, however, the overhead of data movement between CPU and GPU can be a main performance bottleneck, since multiple GPU kernels are invoked across different callsites, resulting in complex communication patterns between CPU and GPU. For those applications, reducing redundant data transfers is critical for optimal performance. In the R-Stream model, to globally optimize the data movement across different mappable parallel regions, these regions should be put together into one function. However, if any unmappable code exists between the mappable regions, the R-Stream compiler cannot handle the resulting function. To deal with this issue, the R-Stream model provides a technique, called *blackboxing*, to mask unmappable code. In the current implementation, however, the blackboxing technique is not yet fully supported for porting to GPUs.

The R-Stream model allows a user to control various loop transformations and communication optimizations for each mappable region, but not across different regions. To optimize the data transfers across different mappable regions, these regions should be merged into one function, as explained above. However, merging multiple regions may increase the mapping complexity of the underlying polyhedral model to the level that cannot be handled by the current R-Stream compiler. For the same reason, there is a practical limit on the maximum depth of nested loops.

IV. Evaluation Methodology

For our evaluation, we selected thirteen OpenMP programs to port and optimize for CUDA GPUs using each of the programming models explained in Section III. These programs are two kernel benchmarks (*JACOBI* and *SPMUL*), three NAS OpenMP Parallel Benchmarks (*EP*, *CG*, and *FT*), and eight Rodinia Benchmarks [19] (*BACKPROP*, *BFS*, *CFD*, *SRAD*, *HOTSPOT*, *KMEANS*, *LUD*, and *NW*).

Tested programs include both regular and irregular applications in diverse domains. Some of these programs are not considered 'GPU-friendly,' but the diversity allows us to gain valuable insights on the application coverage. For programming models not accepting OpenMP directives, all OpenMP directives and library calls were either converted to their corresponding directives or removed from the input OpenMP programs. To achieve the best performance, additional directives were added for each programming model, and code structures of the input programs were also modified to meet the requirements and suggestions of each model. More details on the code changes of each program are explained in the following sections.

We used a NVIDIA Tesla M2090 GPU as an experimental platform. The device has 512 streaming processors with clock rate at 1.3 GHz and has 6GB DRAM. The host system–Keeneland [1]–consists of eight Intel Xeon X5660 CPUs at 2.8 GHz. Performance of each programming model was evaluated by measuring the speedups of output CUDA programs generated by each programming model. The speedups are over sequential CPU versions without OpenMP, compiled with the GCC compiler version 4.1.2 using option *-O3*. For CUDA compilation, NVIDIA Toolkit version 4.1 was used. For both PGI Accelerator and OpenACC, PGI compiler version 12.6 was used, and for R-Stream, HMPP, and OpenMPC, version 3.2RC1, 3.0.7, and 0.31 were used respectively. To compare against GPU performance upper bound, the hand-written CUDA versions of the tested programs were also evaluated; Rodinia Benchmarks have both OpenMP and CUDA versions, and thus CUDA versions in the benchmarks suite were used. The CUDA version of *FT* is from Hpcgpu Project [20], and for the remaining ones (*JACOBI*, *SPMUL*, *EP*, and *CG*), we created hand-tuned CUDA versions by converting from the OpenMP versions manually.

TABLE II
PROGRAM COVERAGE AND NORMALIZED, AVERAGE CODE-SIZE INCREASE. THE COVERAGE REPRESENTS THE PERCENTAGE OF OPENMP PARALLEL REGIONS TRANSLATED TO GPU KERNELS BY EACH MODEL (*A/B* FORMAT INDICATES THAT TESTED MODELS CAN CONVERT *A* OUT OF *B* OPENMP PARALLEL REGIONS INTO GPU CODES.) THE NORMALIZED CODE-SIZE INCREASE INDICATES HOW MUCH ADDITIONAL CODES, ON AVERAGE, SHOULD BE ADDED TO PORT THE INPUT OPENMP PROGRAMS INTO GPU USING EACH MODEL.

GPU Models	Program Coverage (%)	Code-Size Increase (%)
PGI Accelerator	98.3 (57/58)	18.2
OpenACC	98.3 (57/58)	18
HMPP	98.3 (57/58)	18.5
OpenMPC	100 (58/58)	5.2
R-Stream	37.9 (22/58)	9.5

V. Experimental Results

Table II summarizes the program coverage and normalized code-size increase of each GPU programming model. The program coverage suggests that most of the existing GPU models can translate computation patterns from various application domains. The R-Stream model has relatively low coverage, since it works only on regular affine programs. If its code-masking mechanism, called *blackboxing*, is properly used, its coverage may increase to the level similar to other models, but the current R-Stream compiler does not support the feature yet. The code-size increase refers to the normalized, average amount of additional codes that are needed to conform to each programming model and to manually optimize data transfers between CPU and GPU. Even though the R-Stream model provides the highest level of abstraction, the required coding effort is not the smallest; to mask irregular codes, they should be outlined as separate

functions, and dummy affine functions, which summarize array access patterns of the original irregular codes, should be created to replace the outlined functions. The OpenMPC model has the least amount of code restructuring, since it supports function calls in GPU codes and performs selective procedure cloning automatically, which are often necessary for context-sensitive, global data-transfer optimizations; other models either do not support these features or support them in limited ways, and thus these models often request manual inlining/procedure cloning in the input codes. The table shows that HMPP, PGI Accelerator, and OpenACC have similar code-size increases, but required coding practices are different; the HMPP model does not support directives such as *data regions* in PGI Accelerator/OpenACC. Without data regions, the HMPP model requires to explicitly specify data-transfer rules for each codelet for manual optimizations, incurring more directives than PGI Accelerator/OpenACC. However, more explicit control combined with its own runtime support allows HMPP users to express context-sensitive, global data-transfer optimizations with less manual code restructuring than current PGI Accelerator/OpenACC implementations. In the tested benchmarks, these opposite coding practices balance out each other, resulting in similar code increases in these models.

Figure 1 shows the overall performance of GPU programs translated by directive-based GPU compilers (The results for R-Stream are excluded due to its low program coverage, which will be fixed once its masking mechanism becomes available.) Overall, directive-based GPU programming models in the figure achieve similar performance, which are sometimes better than the hand-written CUDA programs. More detailed explanations are presented in the following subsections.

A. Performance of Kernel Benchmarks and NAS Parallel Benchmarks

JACOBI is a widely used kernel constituting the main loop of an iterative method in regular scientific applications. In the original OpenMP version, the outermost loops of nested parallel computation loops are parallelized to minimize parallelization overheads such as OpenMP fork-join overheads. The same rule is applied for GPU computing; parallelizing the outermost loops will incur less GPU kernel invocation overheads than the other way. However, the PGI Accelerator/OpenACC versions following the OpenMP-style parallelization do not perform well. This performance degradation is mostly due to the overhead in large, uncoalesced global memory access patterns. In OpenMPC, the compiler automatically changes the uncoalesced access patterns to coalesced ones by applying a loop-interchange technique called *parallel loop-swap* [21]. In PGI Accelerator/OpenACC, if both outer and inner loops are annotated with *parallel/gang* clauses, the compiler can find a mapping between loop iterations and two-dimensional thread blocks allowing coalesced accesses. Moreover, the PGI compiler automatically applies tiling transformation to exploit GPU shared memory. The manual CUDA version applies the same tiling transformation as the PGI compiler. The best performance of the PGI Accelerator/OpenACC versions is achieved when the *parallel loop-swap* technique is manually applied, and only the outermost loops are parallelized. HMPP version also applies the same techniques, but these transformations can be fully expressed using its directives, since it provides rich set of directives for various loop transformations.

Embarrassingly Parallel (*EP*) is one of NAS OpenMP Parallel Benchmarks. *EP* has abundant parallelism with minimal communication; it is often used to demonstrate the performance of parallel computing systems. To port *EP* using the PGI Accelerator model, several modifications on the code structure are needed; first, *EP* has one parallel region containing an *omp for* loop and *omp critical* section, but the PGI Accelerator compiler cannot parallelize general structured blocks, and it also does not understand the concept of critical sections. Therefore, the parallel region should be converted to an explicit parallel loop. Second, the critical section performs array reduction computation, but the PGI Accelerator model can handle only scalar reductions, and thus the array reduction should be manually decomposed to a set of scalar reductions. Third, *EP* uses a private array to keep internal outputs computed by each thread. In the PGI Accelerator model, the private array is allocated in the GPU global memory for each thread. However, if the number of threads are too big, the allocation of the private array causes a memory overflow. In the PGI Accelerator model, a user can control the size of a thread block indirectly, but the number of thread blocks are implicitly decided based on the size of thread block and the size of loop iteration space. Therefore, to prevent the memory overflow, programmers should manually strip-mine the parallel loop to reduce the size of the loop iteration space. To port using the other models except for OpenMPC, code modifications similar to those for PGI Accelerator were applied.

Figure 1 shows that OpenMPC outperforms the other directive models; the main reason for this performance gap lies in the difference in allocating the private array. All versions use an array expansion technique to privatize an array, but in different manners; the PGI Accelerator compiler expands the array in a row-wise fashion, while the OpenMPC compiler does it in a column-wise fashion using the technique called *Matrix transpose* [21]. The row-wise expansion is a technique commonly used in shared memory systems, since it increases intra-thread locality. However, it causes an uncoalesced memory access problems when applied in the CUDA memory model. On the other hand, column-wise expansion expands an array in a way to increase inter-thread locality, which allows coalesced memory accesses in the CUDA memory model. If the Matrix Transpose technique is manually applied to the other models, instead of using their *private* clauses, they also perform similarly to the OpenMPC version.

The performance gap between the OpenMPC version and the manual CUDA version is due to the difference in handling a *critical* section performing an array reduction; both the OpenMPC version and manual version convert the critical section into array reduction code, but the manual version optimizes further by removing a redundant private array, which was used as a local reduction variable.

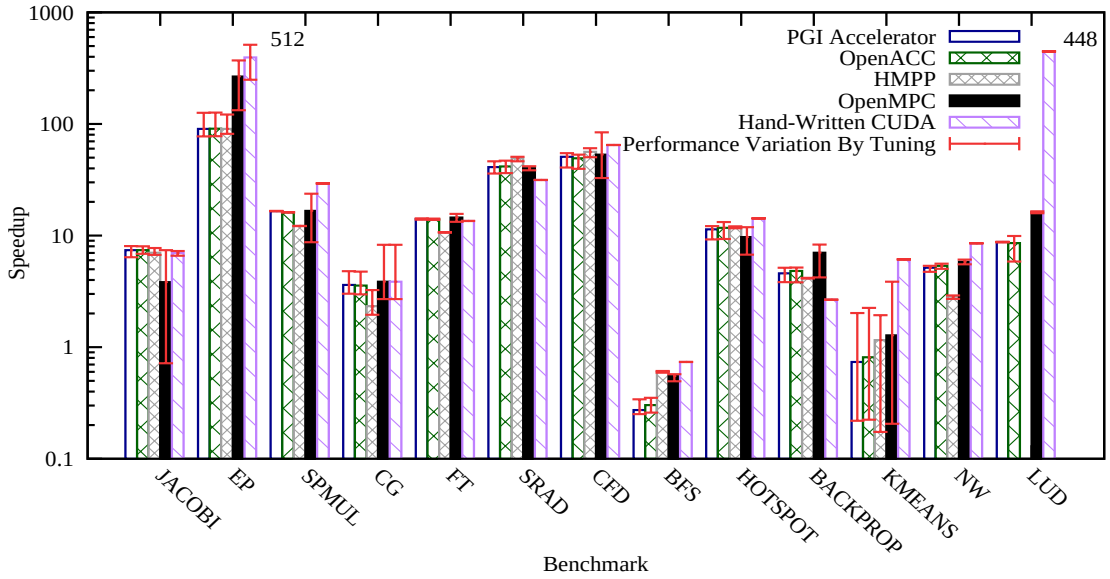

Fig. 1. Performance of GPU programs translated by directive-based GPU compilers. Speedups are over serial on the CPU, when the largest available input data were used

Sparse matrix-vector (SpMV) multiplication is an important representative of irregular applications. The *SPMUL* kernel and the NAS Parallel Benchmark *CG* are two important applications performing the SpMV computations.

While both *CG* and *SPMUL* use similar SpMV multiplication algorithms, *CG* poses additional challenges. In *CG*, many parallel loops span across several procedures, resulting in complex memory transfer patterns between the CPU and GPU. While OpenMPC performs these communication optimizations automatically through its built-in interprocedural data flow analyses, all the other GPU models demand extensive use of data clauses to optimize the complex communication patterns. The OpenMPC version outperforms the other ones, mainly due to a loop transformation technique called *loop collapsing* [21], which addresses uncoalesced access problems caused by indirect accesses and control flow divergences in the CG. Contrary to OpenMPC, the PGI Accelerator compiler uses GPU shared memory extensively in order to alleviate the uncoalesced memory access problems.

The NAS OpenMP Parallel Benchmark *FT* solves a 3-D partial differential equation using the Fast Fourier Transform (FFT). The original OpenMP version heavily uses data blocking to exploit intra-thread locality on traditional, cache-based systems. Moreover, for the best intra-thread locality, nested loops performing FFT are parallelized across 3rd or 2nd dimensions, instead of 1st dimension. However, this data partitioning scheme allows little opportunity for coalesced global memory accesses.

The hand-written CUDA version applies a different data partitioning scheme; it transposes the whole 3-D matrix so that 1st dimension is always parallelized for all 1-D FFT computations. It also linearizes all 2-D and 3-D arrays to reduce memory-address-computation overhead. These changes may not be easy to be automated by compilers, but can be applied to the input OpenMP code. After the input code modifications, all the GPU models achieve the performance comparable to the hand-written CUDA version.

B. Performance of Rodinia Benchmarks

SRAD performs a diffusion method called Speckle Reducing Anisotropic Diffusion (SRAD) to remove locally correlated noise in ultrasonic and radar imaging applications based on partial differential equations. The OpenMPC version uses a *parallel loop-swap* optimization to achieve the coalesced accesses, while the other models rely on multi-dimensional loop partitioning, as in the manual version. The manual version applies an additional optimization to reduce the number of global memory accesses; the original OpenMP version uses subscript arrays to store index expressions for subscripted arrays, but the manual version calculates the index expressions directly to remove global memory accesses to read the subscript arrays. However, this optimization increases the amount of computations and control flow divergences for calculating the index expressions. The experimental results reveal that the performance gain by reducing global memory accesses are overwhelmed by the overhead of additional control flow divergences.

Breadth-First Search (*BFS*) is one of fundamental graph algorithms widely used in many scientific and engineering applications. Even though it has a very simple algorithm, its irregular access patterns using a subscript array make it difficult to achieve performance on the GPU. Therefore, none of tested models achieved reasonable performance. There exist a GPU implementation [22] performing better than classical CPU implementation. However, the algorithms in the GPU implementation are not expressible in directive-based models.

CFD is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. The

naive translation using directive models has some speedups but much less than the manual version. The performance gap is largely due to the uncoalesced global memory accesses. *CFD* uses several two-dimensional matrix data, and accessing these matrices causes uncoalesced memory access problems. However, it is very difficult for a compiler to fix the problem automatically, since the matrices are stored in one-dimensional arrays and accessed using complex subscript expressions. The manual version modifies the layout of these two-dimensional data stored in one-dimensional arrays and changes the corresponding array subscript expressions in a way to enable coalesced memory accesses. After these changes were manually applied to the input OpenMP codes, GPU models could achieve similar performance to the manual CUDA version. The OpenMPC version performs better than the other ones mainly due to more fine-grained data caching to exploit both constant cache and texture cache.

HOTSPOT is a tool to estimate processor temperature based on an architectural floorplan and simulated power measurements. The original OpenMP program contains two parallel loops, and each loop has a nested inner loop, which is also parallelizable. Parallelizing the outer loops as expressed in the original OpenMP program does not provide enough number of threads to hide the global memory latency. The manual version uses a two-dimensional partitioning scheme to increase the number of threads. It also use tiling transformation to exploit shared memory. The OpenMPC model does not support multi-dimensional partitioning scheme. Rather, it uses the OpenMP *collapse* clause to produce a similar effect. Other GPU models used manual collapsing instead of multi-dimensioning partitioning, since some necessary features are not yet implemented.

Back Propagation (*BACKPROP*) is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. In OpenMPC, naive translation of *BACKPROP* performs very poorly, due to uncoalesced accesses. In fact, the *parallel loop-swap* technique can fix the uncoalesced global memory access problems. However, the current OpenMPC compiler could not perform the optimization automatically due to its complexity. Therefore, the technique had to be applied manually.

In other models, code transformations similar to OpenMPC version were applied. Moreover, to avoid array reduction, which occurs as a side-effect of the memory layout change but cannot be handled by these models, some nested parallel loops had to be manually transformed further.

K-Means (*KMEANS*) is a clustering algorithm used extensively in data-mining and many other areas. It has reduction patterns, but the original OpenMP version does not use OpenMP *reduction* clauses, since OpenMP does not support array reductions. Instead, the OpenMP version uses expanded arrays for each thread to hold its partial output, and the final reduction is performed by the CPU. Most GPU models use the original pattern, but for OpenMPC, the reduction patterns were changed to OpenMP *critical* sections so that the OpenMPC compiler can recognize them as reduction operations, resulting better performance than other models. Both the OpenMPC version and the manual CUDA version use the similar two-level tree reduction forms [23]. However, the manual version performs much better than the OpenMPC version; this performance gap is mainly due to the differences in implementing the two-level reductions. The manual version applies a complex subscript manipulation to reduce partial reduction output size so that it can be cached in the GPU shared memory. To express these changes in the existing directive models, however, special extensions to explicitly express the shared memory and GPU thread IDs will be needed.

Needleman-Wunsch (*NW*) is a nonlinear global optimization method for DNA sequence alignments. To achieve the optimal GPU performance, a tiling optimization using shared memory is essential. Due to the boundary access patterns, however, our tested compilers could not generate efficient tiling codes, which attribute to the main reason for the performance gap between manual version and automatic versions.

LU decomposition (*LUD*) is a matrix decomposition tool. In the original OpenMP version, the main computation consists of only two simple parallel loops. However, it is known to be very difficult for compilers to analyze and generate efficient GPU code, due to its unique access patterns. The hand-written CUDA code shows that algorithmic changes specialized for the underlying GPU memory model can change its performance by an order of magnitude. Unfortunately, we could not express the complex manual optimizations in the input code, and thus GPU models performs poorly compared to the manual CUDA version. This large performance gap motivates that for the best performance, the high-level translation system also needs to support an alternative way to express the CUDA programming and memory model in high level.

VI. Current Issues and Future Directions

Combining qualitative analysis in Section III and the actual evaluation on various application domains in Section V gives us valuable insights to identify various issues and possible research directions, some of which are described in the following subsections.

A. Functionality

The existing models target structured blocks with parallel loops for offloading to a GPU device, but there exist several limits on their applicability.

1) Most of the existing implementations work only on loops, but not on general structured blocks, such as OpenMP's parallel regions.
2) Most of them either do not provide reduction clauses or can handle only scalar reductions.
3) None of existing models supports critical sections or atomic operations (OpenMPC accepts OpenMP critical sections, but only if they have reduction patterns.).
4) All existing models support synchronization in limited ways.

5) Most of the models do not allow function calls in mappable regions, except for simple functions that compilers can inline automatically.
6) All work on array-based computations but support pointer operations in very limited ways.
7) Some compilers have limits on the complexity of mappable regions, such as the shape and depth of nested loops and memory allocation patterns.

Because each model has different limits on its applicability, the programming effort required to conform to their models also differs across models, leading to portability problems. OpenACC is the first step toward a single standard for directive-based GPU programming, but several technical details are remain unspecified. Therefore, continued, in-depth evaluations and research on these models is essential.

B. Scalability

All existing models assume *host+accelerator* systems where one or small number of GPUs are attached to the host CPU. However, these models will be applicable only to small scale. To program systems consisting of clusters of GPUs, hybrid approaches such as *MPI + X* will be needed. To enable seamless porting to large scale systems, research on unified, directive-based programming models, which integrate data distribution, parallelization, synchronization, and other additional features, will be needed. This research will work as a basis to solve the grand Exascale computing challenges. Users will also need tools and techniques to optimize the interactions between MPI and these programming models [24].

C. Tunability

Tuning GPU programs is known to be difficult, due to the complex relationship between programs and performance. Directive-based GPU programming models may enable an easy tuning environment that assists users in generating GPU programs in many optimization variants without detailed knowledge of the complex GPU programming and memory models. However, most of the existing models do not provide enough control over various compiler-optimizations and GPU-specific features, posing a limit on their tunability. To achieve the best performance on some applications, research on alternative, but still high-level interface to express GPU-specific programming model and memory model will be necessary.

D. Debuggability

The high-level abstraction offered by directive models puts a significant burden on the user in terms of debugging. The existing models do not provide insight to the users on the multiple levels of translation, and some implementations either do not always observe directives inserted by programmers, or worse, they translate them incorrectly. In particular cases, if user-provided directives conflict with the compiler's internal analyses, the compiler might generate incorrect code without warning. This aspect of the problem adds a significant challenge to this already complex debugging space. For debugging purpose, all existing models can generate CUDA codes as intermediate output, but most of existing compilers generate CUDA codes by unparsing low-level intermediate representation (IR), which contain implementation-specific code structures and thus are very difficult to understand. More research on traceability mechanisms and high-level IR-based compiler-transformation techniques will be needed for better debuggability.

VII. Conclusion

In this paper, we have performed an early evaluation of new, directive-based, high-level GPU programming models. The directive-based GPU programming models provide very high-level abstraction on GPU programming, since the designated compilers hide most of the complex details specific to the underlying GPU architectures. In our evaluation of existing directive-based GPU programming models, we ported and optimized thirteen OpenMP applications from various scientific domains to CUDA GPUs. Our results show that these models have made great strides over the past two years, and can achieve reasonable performance, even though they differ in the levels of abstraction and programming complexity to conform to the models and provide optimized performance. Additionally, as we see changes in future architectures that combine GPUs and CPUs onto the same die [25], we will need these higher levels of abstraction and requisite compilation tools to provide performance portability, isolating applications from dramatic differences in their architectural components.

However, the differences in their expressibility and the various constraints in the underlying compiler implementations also lead to more detailed questions on the requirements for standard GPU programming models and right scope of the interaction between programming models and the compilers and runtime systems.

Our detailed comparison with the corresponding hand-written CUDA programs reveals that the current high-level GPU programming models may need to be further extended to support alternative, high-level methods and optimizations in order to express architecture-specific details and memory models in order to achieve the competitive performance in some applications.

Acknowledgment

The paper has been authored by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract #DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research is sponsored in part by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy and National Science Foundation award OCI-0910735.

References

[1] J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing heterogeneous gpu computing to the computational science community," *IEEE Computing in Science and Engineering*, vol. 13, no. 5, pp. 90–95, 2011.

[2] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-. Andre, D. Barkai, J.-. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, RobertHarrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Y. Ishikawa, Z. Jin, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. Mueller, W. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The International Exascale Software Project RoadMap," *Journal of High Performance Computer Applications*, vol. 25, no. 1, 2011.

[3] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balaji, P. C. Diniz, A. Koniges, M. Snir, and S. R. Sachs, "Report of the 2011 workshop on exascale programming challenges," US Department of Energy, Tech. Rep., 2011.

[4] Sh, "Sh: A metaprogramming language for programmable GPUs. [online]. available: http://www.libsh.org," (accessed April 02, 2012).

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004, pp. 777–786.

[6] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and computation transformations for brook streaming applications on multiprocessors," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 196–207.

[7] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for GPUs," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*. New York, NY, USA: ACM, 2006, p. 184.

[8] CUDA, "NVIDIA CUDA [online]. available: http://developer.nvidia.com/category/zone/cuda-zone," 2012, (accessed April 02, 2012).

[9] OpenCL, "OpenCL [Online]. Available: http://www.khronos.org/opencl/," 2012, (accessed April 02, 2012).

[10] OpenMP, "OpenMP [Online]. Available: http://openmp.org/wp/," 2012, (accessed April 02, 2012).

[11] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, 2011.

[12] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE press, 2010.

[13] PGI_Accelerator, "The Portland Group, PGI Fortran and C Accelarator Programming Model [Online]. Available: http://www.pgroup.com/resources/accel.htm," 2009, (accessed April 02, 2012).

[14] HMPP, "HMPP Workbench, a directive-based compiler for hybrid computing [Online]. Available: www.caps-entreprise.com/hmpp.html," 2009, (accessed April 02, 2012).

[15] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 51–61.

[16] OpenACC, "OpenACC: Directives for Accelerators [Online]. Available: http://www.openacc-standard.org," 2011, (accessed April 02, 2012).

[17] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "OpenMP for Accelerators." in *IWOMP'11*, 2011, pp. 108–121.

[18] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, and R. Graham, "Experiences with High-Level Programming Directives for Porting Applications to GPUs," in *Facing the Multicore - Challenge II*. Springer Berlin Heidelberg, 2012, pp. 96–107.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. ha Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[20] L. L. Pilla, "Hpcgpu Project [Online]. Available: http://hpcgpu.codeplex.com/," 2012, (accessed April 02, 2012).

[21] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: ACM, Feb. 2009, pp. 101–110.

[22] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 52–55.

[23] CUDA_reduction, "NVIDIA CUDA SDK - CUDA Parallel Reduction [online]. available: http://developer.nvidia.com/cuda-cc-sdk-code-samples#reduction," 2012, (accessed April 02, 2012).

[24] J. S. Meredith, P. C. Roth, K. L. Spafford, and J. S. Vetter, "Performance implications of nonuniform device topologies in scalable heterogeneous architectures," *IEEE Micro*, vol. 31, no. 5, pp. 66–75, 2011. [Online]. Available: http://dx.doi.org/10.1109/MM.2011.79

[25] K. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, "The tradeoffs of fused memory hierarchies in heterogeneous architectures," in *ACM Computing Frontiers (CF)*. Cagliari, Italy: ACM, 2012.